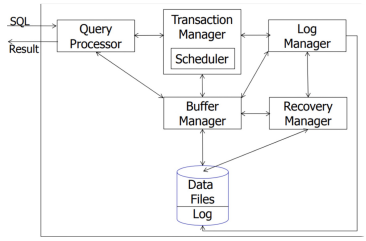


Dense: has atleast 1 instance from every value appearing in the column.

Sparse: records apear for some records not all.

Primary Index (dense):
→ Sorted on the PK
→ Can have only 1 primary Index
+ Fast
− more space

Primary Index (Sparse):
→ Sorted on the PK
→ Can have only 1 primary Index
+ less space, more efficient
- slower

Secondary Index:
→ Created from a Candidate key not the primary key
→ First level must be dense

Multi-Level Index: Index on an Index

Block Range Index (BRIN):
→ type of sparse
→ Min/Max stored in index for every page
→ allows skipping large sections of table
→ Ideal for column used in sorting

When to use? O(log r) r = no of ranges
→ Doing analytical queries (sum, avg ..)
→ lots of search with small range
→ exact value or range on a very small % of stored data in table

B+ tree: O(log n)

1- Search:
1 Value: Follow the pointers
range: Start from LB, stop at UB
value < UB: start left most till UB
LB < value: start LB till right most

2- Insert:
if (room available)
Insert
else
→ Split node to 2, insert
→ Adjust parent node

	Max ptrs	Max keys	Min ptrs	Min keys
Non-leaf (non-root)	n+1	n	⌈(n+1)/2⌉	⌈(n+1)/2⌉ − 1
Leaf (non-root)	n+1	n	⌊(n+1)/2⌋	⌊(n+1)/2⌋
Root	n+1	n	1	1

3- Delete:
→ Find the key, remove it
→ Delete pointer
if (Min still true)
Done
else
Search for Sibling node
Above Min: take extra key from sibling, Adjust parent
= Min: Merge, adjust delete at Parent

When to use?
→ query Single term On exact value
→ query Range Search
→ query with agg func

When not to use?
→ query returning all/most of data

Spatial data:
→ Describes location, area info, shape
→ represented as points/lines/polygons

Types:
Point data: Points in a multi-dimension space
Region data: Objects have spatial extent with location & Boundary

When to use?:
Data Spatially close to each other

R- tree:
→ Organize data into rectangles for fast Search
→ Identified by upper left, bottom right

Insert:
to rectangle with Smallest area inc if all Full Split

Delete:
new smaller rectangle, merge if too low

Hashing: Search O(1)
Hash table → Buckets/Blocks → records (contain)
Bucket address → Insert key to hash func
util = used / all (Don't add Overflow)

Static hashing:
Fixed no of buckets
Insert: if bucket full add Overflow bucket
Delete: Delete then Shift up
keep util between 50 & 80
< 50 Wasting space
> 80 Overflow significant

Dynamic hashing:

1- Extensible:
key = Sequence of Bits
i = no of bits to identify memory [left]
j = no of bits to identify bucket

Insert:
Space → insert
Full → j < i: split & adjust; j = i: increment i by 1
+ Can handle growing files
− Doubles in size

2- Linear:
i = no of bits/sequence [right]
insert:
check space → Insert; → no Space, overflow
redirect = Old − 2^{i-1}
Util > threshold: increase buckets by 1
use when: query with single term On exact value
Don't use: returning all data range Search/agg Funcs

Multi key indices:
→ Index on more than one column

Partition hash function: ★★★
→ Concatenate all Outputs from each hash function
+ O(1) + k k checks on entities same cell
+ Can answer partial queries
- nearest neighbor query not possible

Grid Index:
→ In each dimension, values of key are stored
→ if the keys are too much use range
+ handles exact values
+ multi key search O(log m) + k
− needs ranges to split keys

	160	200	250	300
512	{1003}		{1002, 1005}	
1024	{1002}	{1007}	{1001, 1004, 1009}	{1006}
2048	{1011}		{1008}	{1010}

When to use:
→ query with multiple terms on exact values
→ Partial query on exact values
→ aggregate func.
→ nearest neighbour

not to use:
heavy DML (update, insert, delete)

kD tree:
→ Binary tree (Max 2 children)
→ leafs points to buckets
+ O(log n) access
- partial query & nearest neighbour are in efficient

When to use:
→ query with multiple terms on exact values
→ nearest neighbour
→ Small memory foot print

Bitmap Index:
→ Collection of a bit-vector of length n = no of rows
+ answer bitwise operation
- long bits sequence can be solved by Compression of bits

Quad tree:
→ A rooted tree where each node has exactly 4 children

Search:
Start at root, examine each child & check if it intersects the area being searched for. if it does repeat.

Insert:
Start at root determine which quadrant our point occupies, repeat till leaf
Insert.
if full split

Delete:
remove leaf node.
if parent has only 1 child replace the parent node with child leaf

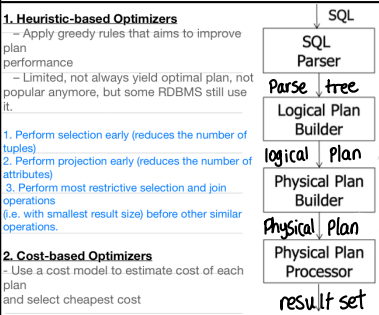
When to use:
→ query with multiple terms on exact values
→ Partial query on exact values
→ aggregate func.
→ nearest neighbour

Don't use:
→ high dimensionality clustered data

1. Heuristic-based Optimizers
– Apply greedy rules that aims to improve plan performance
– Limited, not always yield optimal plan, not popular anymore, but some RDBMS still use it.

1. Perform selection early (reduces the number of tuples)
2. Perform projection early (reduces the number of attributes)
3. Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

2. Cost-based Optimizers
- Use a cost model to estimate cost of each plan and select cheapest cost



Size estimates:
T(R) = # tuples in R
S(R) = # bytes in
B(R) = # of pages to hold
V(R,A) = # Distinct Value of A

$W = R_1 \times R_2$
$T(W) = R_1 \times R_2$
$S(W) = R_1 + R_2$

$W = \sigma_{A=a}(R)$
$S(W) = S(R)$
$T(W) = \frac{T(R)}{V(R,A)}$ or $\frac{T(R)}{Dom(R,A)}$

$W = X \bowtie_{x.c=y.c} Y$
$T(W) = T(X)\,T(Y)$

$W = \sigma_{z \geq val}(R)$
$T(W) = \frac{T(R)}{2}$ No Dups
$T(W) = \frac{T(R)}{3}$ Dups

$T(W) = T(R) * \frac{max - val + 1}{max - min + 1}$
(values in Range)

$W = \sigma_{a != val}(R)$
$T(W) \simeq T(R)$

$W = \delta_A(R)$ Distinct
$T(W) = \frac{1}{2} T(R)$

$W = R_1 \bowtie R_2$
$S(W) = S(R_1) + S(R_2) - S(A)$
$T(W) = T(R_1) \times T(R_2)$ $X \cap Y = \emptyset$
$T(W) = \frac{T(R_1) \times T(R_2)}{max(V(R_1,A), V(R_2,A))}$ $X \cap Y = a$
$T(W) = \frac{T(R_1) \times T(R_2)}{Dom(R_2,A)}$ low Fy Domain

SC(R,A)

$\frac{T(R)}{V(R,A)}$ or $\frac{T(R)}{Dom(R,A)}$

$\sigma_{z \leq val}(R)$

$T(w) = T(R) * \frac{val - min + 1}{max - min + 1}$

AND → Multiplication

OR → Addition

Table-scan iterator:

```
Open(){
    b <- first block of R;
    t <- first tuple of b;
}

close(){
}
GetNext(){
  if( t is beyond the last tuple in b)
          increment b;
          if (b is beyond last block)
                return NoMoreData;
          else
                t <- first tuple of b;
  oldt <- t;
  increment t;
  return oldt;
}
```

Bag Union:

open(){ R.open(); curRel ← R; }

close(){ R.close(); S.close(); }

```
  GetNext(){
     if(CurRel = R) {
             t <- R.GetNext();
             if( t != NoMoreData)
                   return t;
             else { // R is exhausted
                   S.Open( );
                   CurRel <- S;
             }
     }
        else return S.GetNext;
  }
```

Set difference:

```
open(){
    R.open();
    S.open();
}

getNext(){
    x = S.getNext( );
    y = R.getNext( );

    while( true ){
        // nothing left in R, so return from S
        if (y == NoMoreData) {
            return x;
        }

        if (x == NoMoreData){
            return NoMoreData;
        }

        // exist in R, move to next tuple in S
        if ( x == y ) {
            x = S.getNext( );
        } else {
            // does not exist in R
            if (x < y) {
                return X;
            } else {
                // might exist in R
                y = R.getNext( );
            }

        }
    }
}

close( ){
    R.close();
    S.close();
}
```

Projection:

```
open(){
    R.open();
}

getNext(){
    Tuple <- R.getNext();
    if (Tuple != NotFound) {
        Return Tuple;
    }
}

close( ){
    R.close();
}
```

Relation at a time:

- Set Union R U S
- Assuming R is the bigger relation:
  - Read S into memory completely and make it accessible through an in-memory index structure
  - output all tuples of S while reading
  - For each tuple f in R, search if it already exists in S, and output if not.

- Set Intersection R ^ S
- Assuming R is the bigger relation:
  - Read S into memory completely and make it accessible through an in-memory index structure
  - For each tuple f in R, search if it already exists in S, and output if true.

- Set Difference R - S
- Assuming R is the bigger relation:
  - Read S into memory completely and make it accessible through an in-memory index structure
  - For each tuple of R, search if it already exists in S
  - If tuple exists in S, then ignore, else output

- Set Difference S - R
- Assuming R is the bigger relation:
  - Read S into memory completely and make it accessible through an in-memory index structure
  - For each tuple f in R, search if it already exists in S, and delete it from S if it exists
  - output all remaining tuples of S

- Cross Product R x S
- Assuming R is the bigger relation:
  - Read S into memory completely and store it in a buffer.
  - No special data structure is required.
  - For each tuple f in R, combine it with each tuple of S and output the result.

- Natural Join R⋈S
- Assume R(X,Y) and S(Y,Z)
- Assuming R is the bigger relation:
  - Read S into memory completely and store it in a balanced tree index or a hashtable.
  - For each tuple of R, search S to see if a matching tuple exists
  - Output if matching tuple found.

Blocking vs. Non-blocking

1. Projection is not a blocking operator
2. Distinct is not a blocking operator
3. Grouping is a blocking operator
4. Set union is not a blocking operator
5. Set intersection is not a blocking operator
6. Set difference can be blocking or not depending on which relation is read first. For instance when computing R–S, if S is read into M –1 blocks and then each block of R is read, the operation is not blocking. However, if R is read into M – 1 blocks and then each S block is read, the operation is blocking.
7. Bag intersection is not a blocking operator
8. Bag difference can be blocking for the same reason as in 6
9. Product is not a blocking operator
10. Natural join is not a blocking operator

Linear Access Methods:

→ Simplest to implement
→ Most expensive O(n)
→ Scans relation Page by Page row by row

Sort-based Two-pass Algorithm

- Suppose relation R is too big to fit in memory which can accommodate only M blocks of data
- The sorting-based 2 pass algorithm have the following basic structure;

1. Read M blocks of records into memory and sort them
2. Write them back to disk
3. Continue steps 1 and 2 until R is exhausted.
4. Use a merge technique to extract relevant results from all the sorted M-blocks on disk

2) Sort-based Algo. For Duplicate Elimination

1. Let relation R, in which duplicates have to be eliminated, be too big to fit in memory
2. Read M << R blocks into memory and sort them.
3. Store the sorted set of M blocks back on disk
4. Take one block from each sorted sub-list on disk and eliminate duplicate occurrences of tuples
   - i.e. take the first element p of the first block and move all other blocks to go beyond p.

Hash based Algorithms

- Basic Idea:
  1. Read relation R block by block
  2. Take each tuple in a block and hash it to one of a set of buckets of a hash file
  3. All "similar" tuples should hash to the same bucket
  4. Examines each bucket in isolation to produce final result.

1) Hash-based Duplicate Elimination

- Eliminate duplicates from relation R that is too big to fit in memory:
  1. Read relation R block by block
  2. Take each tuple in a block and hash it to one of a set of buckets of a hash file.
  3. Visit each bucket and eliminate duplicates using either a two-pass sort based algorithm (if bucket too big)

2) The Hash-Join Algorithm

- Natural Join over relations R(X,Y) and S(Y,Z) is achieved by hashing using just the Y component of R and S.
- All tuples having the same value of the Y component should hash to the same bucket
- If same hash function is used for R and S, corresponding buckets can be compared.
- Use a one-pass join algorithm by using corresponding buckets from the hash files of R and S.

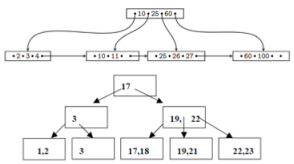
Index Based Algorithms

- Useful when tuples have to be extracted from relations based on attributes that have been indexed
- Indexed-based selection

1) Zig-zag Join

- R(X,Y) * S(Y,Z)
- On 2 B+ trees together
- At best: Read one page from B+ $tree_1$ followed by one page from B+$tree_2$
- Use intermediate level Values to learn about boundaries (skip if needed)
- Output common

- R(X,Y) * S(Y,Z)
- On 2 B+ trees together



Exact = { B+, Ext hash, BRIN, Partion Hash, Bitmap, linear Hash, grid, Quad, R, kD }

Range = { R, Quad, kD, B+, Ext hash, grid, linear hash, Partioned hash, Brin, Bit }

Partial = { Brin, B+, Ext hash, Part hash, grid, linear hash, Quad, R, kD }

Spatial = { R, Q, kD, grid, B+, Ext hash, Part hash, Brin, linear hash, Bit }

Nearest Neigh = { R, Q, kD, grid, B+, Ext hash, Part hash, Brin, Bit, linear hash }